Cours 3 – Traits impératifs

Programmation fonctionnelle
CFA INSTA - Master 1 - Analyste Développeur

Pierre Talbot (pierre.talbot@univ-nantes.fr)

Université de Nantes

10 avril 2019

UNIVERSITÉ DE NANTES

Le menu

Traits impératifs en OCaml (K_4)

$\langle D \rangle$ ::=
- | ... — **Déclaration de types** D_4 (D_3)
- | `exception` **e** `of` T — (exception)

Une instruction impérative est une expression dont le type est `unit`.

$\langle n, m, p, q \ldots \rangle$ ::= — **Expressions**
- | ... — (K_1, K_2, K_3)
- | `raise` n — (lancer une exception)
- | `try` p `with` **e** `->` q — (récupérer une exception)
- | p `;` q — (séquence)
- | `for` **i** `=` n [`to` | `downto`] m `do` p `done` — (boucle `for`)
- | `while` n `do` p `done` — (boucle `while`)
- | **e** `<-` n — (opérateur de mise-à-jour)

Le menu

Exception

```
exception EmptyList

let head = function
  | [] -> raise EmptyList
  | a::_ -> a

let _ =
  try
    head []
  with EmptyList -> Printf.printf "empty list\n"
```

Note : Les exceptions sont très rapides en OCaml et sont souvent utilisées pour sortir d'une récursion (*a contrario* des autres langages (e.g. Java) où une exception signifie un comportement exceptionnel).

Le menu

Séquence

On peut séquencer des instructions retournant `unit` avec l'opérateur de séquence `;`.
Au lieu de :

```
let _ = Printf.printf "hello" in
let _ = Printf.printf " world" in
()
```

On peut écrire :

```
Printf.printf "hello" ;
Printf.printf " world"
```

Séquence : attention...

Un problème très fréquent quand on utilise des ; est d'écrire :

```
let print_hello name =
  Printf.printf "hello";
  Printf.printf " %s\n" name;

let _ = print_hello "you"
```

Séquence : attention...

Un problème très fréquent quand on utilise des ; est d'écrire :

```
let print_hello name =
  Printf.printf "hello";
  Printf.printf " %s\n" name;

let _ = print_hello "you"
```

qui sera compris par le compilateur comme :

```
let print_hello name =
  Printf.printf "hello";
  Printf.printf " %s\n" name;
  let _ = print_hello "you"
```

et génère des erreurs syntaxiques infernales...

Séquence : attention...

La solution est de mettre entre parenthèse systématiquement un groupe d'instructions séquencées :

```
let print_hello name =
  (Printf.printf "hello";
  Printf.printf " %s\n" name;)

let _ = print_hello "you"
```

Ou plus élégamment avec `begin` *p* `end` :

```
let print_hello name =
begin
  Printf.printf "hello";
  Printf.printf " %s\n" name;
end
```

Le menu

Tableau

- Sous forme de bibliothèque :
  https://caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html
- Quelques fonctions principales :

```
val get : 'a array -> int -> 'a
val set : 'a array -> int -> 'a -> unit
val make : int -> 'a -> 'a array
```

Sucre syntaxique pour tableau

Un sucre syntaxique pour créer des tableaux de tailles connues à la compilation :

```
let t = [| 1; 1; 4 |]
```

Faciliter la manipulation de tableau :

```
Array.get t i      →   t.(i)
Array.set t i v    →   t.(i) <- v
```

Boucle

Le corps de la boucle doit avoir le type `unit`.

```
for i=10 downto 0 do
  Printf.printf "%d " i
done
```

En coopération avec les tableaux :

```
let t = [| 1; 4; 2 |] in
for i=0 downto 1 do
  t.(i) <- t.(i) + 1
done
```

Notez que l'ancienne valeur du tableau sera perdue (*in-place modification*).

En bonus : Tableau persistant

Les tableaux persistants :

- Temps d'accès aux éléments constant.
- Utilisation du tableau comme une structure fonctionnelle.
- Parfait pour des algorithmes de *backtracking*.

Voir "Semi-persistent data structures", Sylvain Conchon and Jean-Christophe Filliâtre, 2008.

Le menu

Enregistrement mutable

On peut déclarer certains champs d'un enregistrement comme `mutable`.

```
type point2D = { mutable xy : coord; c: color}
```

La mise à jour d'un champ mutable se fait avec l'opérateur <- comme pour les tableaux :

```
let p = { xy=(make_coord 2 3); c=Red } in
begin
  p <- (make_coord 4 3);
  Printf.printf "(%d %d)" p.xy.x p.xy.y
end
```

Variables impératives

On peut simuler des variables impératives (à état) avec les enregistrements mutables :

```
type cell = {mutable content: int}
```

Ainsi on peut transformer un programme écrit en C de la sorte :

```
int i = 0; i = i + 1
```

$\rightarrow$

```
let i = {content=0} in
i.content <- (i.content + 1)
```

Sucre syntaxique pour références

- Les variables impératives deviennent fréquentes dès qu'on écrit de manière impérative.
- Il existe les références qui sont un sucre syntaxique pour cet enregistrement à un champ mutable.

Le type est noté :

```
type 'a ref = {mutable contents:'a}
```

Et peut être manipulé avec la syntaxe suivante :

```
ref: 'a -> 'a ref
(!): 'a ref -> 'a
(:=): 'a ref -> 'a -> unit
```

Une bonne vieille boucle `while`

En souvenir du temps où vous programmiez en impératif....

```
let i = ref 0 in
while !i < 10 do
  Printf.printf "%d " !i;
  i := !i + 1
done
```

Le menu

Égalité structurelle VS égalité physique

OCaml possède deux types d'égalités :

- = qui est l'égalité structurelle (on compare le contenu).
- == qui est l'égalité physique (on compare l'adresse des éléments).

```
let x = (1, 2) in
let y = x in
let z = (1, 2) in
begin
  if x = y then Printf.printf "x = y";
  if x == y then Printf.printf "x == y";
  if x = z then Printf.printf "x = z";
  if x == z then Printf.printf "x == z";
end
```

Normalement, vous utiliserez toujours = : pour le moment ne vous posez même pas la question.

Le menu

- Traits impératifs
- Conclusion

Conclusion

Le paradigme impératif est présent en OCaml : exception, tableau, boucle et enregistrement mutable.

Points à retenir

- On préférera utiliser par défaut les structures fonctionnelles car plus élégantes et facile à programmer en OCaml.
- Parfois les structures impératives sont nécessaires, notamment pour un accès en temps constant à un élément.
- Généralement, on croit qu'on a besoin d'un tableau, mais les listes sont aussi (voir plus) adaptées (`fold_left`, `map`).